LAWRENCE
LIVERMORE
NATIONAL
LABORATORY

UCRL-JC-153186

Object-Oriented NeuroSys: Parallel Programs for Simulating Large Networks of Biologically Accurate Neurons

P. Pacheco, P. Miller, J. Kim, T. Leese, and Y. Zabiyaka

May 7, 2003

10th European Parallel Virtual Machine/Message Passing Interface Conference, Venice, Italy, September 29 – October 2, 2003

Object-Oriented NeuroSys: Parallel Programs for Simulating Large Networks of Biologically Accurate Neurons *

Peter Pacheco[1], Patrick Miller[2],
Jin Kim[1], Taylor Leese[1], and Yuliya Zabiyaka[1]

[1] Keck Cluster Research Group
Department of Computer Science
University of San Francisco
2130 Fulton Street, San Francisco, CA 94117
{peter,jhkim,tleese,yzabiyak}@cs.usfca.edu
[2] Lawrence Livermore National Laboratory
7000 East Avenue
Livermore, CA 94550
patmiller@llnl.gov

Abstract. Object-oriented NeuroSys is a collection of programs for simulating very large networks of biologically accurate neurons on distributed memory parallel computers. It includes two principle programs: ooNeuroSys, a parallel program for solving the large systems of ordinary differential equations arising from the interconnected neurons, and Neurondiz, a parallel program for visualizing the results of ooNeuroSys. Both programs are designed to be run on clusters and use the MPI library to obtain parallelism. ooNeuroSys also includes an easy-to-use Python interface. This interface allows neuroscientists to quickly develop and test complex neuron models. Both ooNeuroSys and Neurondiz have a design that allows for both high performance and relative ease of maintenance.

Keywords: parallel computing, parallel CVODE, parallel visualization, Python, computational neuroscience.

1 Introduction

One of the most important problems in computational neuroscience is understanding how large populations of neurons represent, store and process information. Object-oriented NeuroSys is a collection of parallel programs that have been designed to provide computational neuroscientists with powerful, easy-to-use tools for the study of networks of millions of biologically accurate neurons. It consists of two principle programs: ooNeuroSys, a program for solving the systems of ordinary differential equations resulting from large networks of interconnected neurons, and Neurondiz, a program for visualizing the results of ooNeuroSys. Both programs use an object-oriented design which makes it relatively easy to add features and change underlying data structures and algorithms. This design has also allowed us to achieve levels of performance better than earlier, structured versions of these programs [12].

In addition to improving performance and maintainability, Object-oriented NeuroSys is much easier to use than the earlier versions. Inter alia, we have completely redesigned the GUI for Neurondiz and we have added a Python [13] interface to ooNeuroSys which allows users to code and test neuron models with relative ease.

The differential equation solver, ooNeuroSys, employs an adaptive communication scheme in which it determines at runtime how to structure interprocess communication. It also makes use of the parallel CVODE library [2] for the solution of the systems of ordinary differential equations.

In this paper we discuss the problems we encountered with an earlier version of NeuroSys and how we decided to address these in ooNeuroSys. We continue with a discussion of the communication schemes we used in ooNeuroSys, the new version of Neurondiz, the Python interface, the performance of Object-oriented NeuroSys, and directions for further development.

Acknowledgements. The first author thanks the W.M. Keck Foundation for a grant to build a cluster for use in research and education and partial support of this research. The work of the first and second authors was performed under the auspices of the U.S. Department of Energy by Lawrence Livermore National Laboratory under contract W-7405-ENG-48.

2 ooNeuroSys Design

The original version of NeuroSys [11, 12] was written in the late 1990's by USF's Applied Mathematics Research Laboratory. It is a very powerful system: we were able to simulate sparsely interconnected systems consisting of 256,000 Hodgkin-Huxley type [4] neurons on a 32-processor cluster with parallel efficiencies of better than 90%. However, its design is essentially structured, and as it grew, it became very difficult to incorporate new and to improve existing features. So in the summer of 2001, we began a complete rebuilding of NeuroSys. One of the central features of the new system is an object-oriented design that makes maintenance relatively easy, but actually improves both performance and scalability.

The original version of NeuroSys was written in C, and it used the MPI library to obtain parallelism. Because of the ease with which the performance and memory-usage of C code can be optimized, we decided to continue to use C for ooNeuroSys. Since our main target platform is clusters of relatively small SMPs, we also continued to use MPI. In our object-oriented design classes are defined in separate C source files. The data members are defined in a struct, and the methods are just C functions whose first argument is a pointer to the struct. Methods can be made private by declaring them to be static. During initial development, underlying data structures are hidden by using incomplete types in header files. That is, a data structure is declared in a dot-c file, and an incomplete type that is a pointer to the struct is declared in the corresponding header file. After initial development — when performance becomes a consideration — underlying data structures are selectively exposed in the header files, and some methods — especially accessor methods — are replaced by macros. A primitive and, for our purposes, completely satisfactory form of inheritance is obtained by declaring members of parent classes in macros in header files that are included in child classes.

Both the original version of NeuroSys and ooNeuroSys take as input two main data structures: a description of the neurons and their interconnections and a list of initial conditions for the variables associated with each neuron. For production simulations these data structures are stored in files and read in at the beginning of program execution. The output of the program is the raw data produced during the simulation, i.e., a collection of variable values as specified by the user.

The original version of NeuroSys used a very simple parallel implementation of a classical, fourth-order Runge-Kutta solver. Although we obtained excellent efficiencies with as many as 32 processors, because of the large amount of communication required, we did not expect this method to scale well for much larger numbers of processors. So in ooNeuroSys we wrote a solver class which interfaces with parallel CVODE [2], a general purpose ordinary differential equation solver for stiff and nonstiff ODE's. It obtains parallelism by partitioning the equations and variables of the system among the parallel processes. Since the equations we solve in ooNeuroSys are nonstiff, we use CVODE's variable-stepsize, variable-order Adams-Moulton solver. CVODE also uses an object-oriented design, and since it was written in C, and it uses the MPI library, writing the solver class interface was completely straightforward.

3 ooNeuroSys Communication

Since the equations and variables are partitioned among the processors, communication is required when an equation depends on a variable assigned to a different process. An interesting feature of neuronal interconnection networks is that the interdependence can have almost any form. At one extreme the problem can be almost perfectly parallel in that the equations assigned to each process have little or no dependence on variables assigned to other processes. At the other

extreme, a problem may be "fully interconnected," i.e., every neuron depends on almost every neuron in the system. Thus the communication requirements of a problem are only known at runtime: after the neuronal interconnection structure has been read in. The original version of NeuroSys partially addressed this by providing *compile-time* options for choosing a communication scheme. This has the obvious weakness that the program needs to be recompiled for different interconnection networks. It has the added weakness that it requires the user to evaluate the interconnection network and then determine which communication program to use. Because of the nature of the problems (thousands or millions of neurons with possibly random interconnects), it may be impossible for a human to evaluate the interconnect, and expecting a neuroscientist to understand the intricacies of interprocess communication schemes may be unreasonable. Thus, ooNeuroSys determines automatically at runtime which communication scheme to use.

Although we have written a variety of communication classes, in the problems that we have studied thus far, we have found that only two of them are necessary for the best performance. The first, which corresponds to a fully interconnected system, is an implementation of `MPI_Allgather` that uses a rotating scheme of send/receive pairs, in which during the ith stage, $i = 0, 1, \ldots$ each process sends all its data to

$$\texttt{my\_rank} + 2^i \bmod p,$$

where p is the number of processes. This results in a communication scheme requiring $\lceil \log_2(p) \rceil$ stages. Its principle complication is that, in general, processes will not be communicating contiguous blocks of data. However, our benchmarks showed that for the data sets we expect to be of interest, both the MPICH [9] and LAM [7] implementations of `MPI_Type_indexed` allowed us to obtain excellent performance on fast-ethernet-connected clusters. We also found that using this implementation with the MPICH-GM [10] implementation resulted in excellent performance on Myrinet-connected clusters.

The second communication scheme is used for sparsely interconnected networks and is essentially that described in [15]. On the basis of the information describing the assigned subnetwork, each process can determine which variables it needs and to which process each variable is assigned. Thus, after a global communication phase, each process knows both which variables it should receive from which processes and which variables it should send to which processes. Once each process has built its communication structure, the interprocess communication is implemented by having each process post nonblocking receives for all of the messages it needs to receive and then making blocking sends of all the messages needed by the other processes. In experiments we found that this "lazy" approach outperformed any heuristic scheduling schemes we devised.

A key feature of the systems modelled by NeuroSys is that if the neuronal interconnection changes at all during a simulation, it usually doesn't change for hundreds or thousands of timesteps. We exploit this by building derived datatypes and persistent requests only at the beginning of a run and when the interconnect changes.

At the beginning of a run, after the neuronal interconnect has been read in, the communication class uses a simple heuristic in order to determine which of the two communication schemes is superior. If the number of processes is small (typically ≤ 4), we use the dense scheme. Otherwise we use a (heuristic) algorithm to construct a deterministic schedule for the sparse communications. If the maximum number of stages required by the schedule is less than $\lceil \log(p) \rceil$, then we expect the sparse scheme will outperform the dense scheme. On the other hand, empirical tests have shown that if the maximum number of stages is more than $\lceil 4 \log(p) \rceil$ in the sparse scheme, then the dense scheme is likely to be superior. For other interconnects, the program runs a short series of benchmarks of each scheme, and chooses the scheme with the smaller overall run time. Since the neuronal interconnection network changes infrequently, if at all, the benchmarks add very little to the overall run time.

4 Neurondiz

Neurondiz takes the output data produced by ooNeuroSys and provides a graphic display of the simulation. In order to handle the large amounts of data produced by ooNeuroSys, Neurondiz has been divided into two components: a backend that reads in and processes the ooNeuroSys data, and a frontend that manages the display. The backend is written in C, and it uses the MPI library. We are working on two versions of the frontend. For less demanding problems, we have a Java frontend. We chose Java to allow for future deployment of a Neurondiz applet, to provide an easy interface to other applications, and to allow for rapid development of new functionalities. For more demanding applications the second version of the frontend uses C++, the Qt application development framework [14] and the OpenGL graphics library [16].

Neurondiz was especially designed to support two hardware configurations. In one configuration, we assume the user is working with a parallel computer with no graphics capability. For example, a user with access to a powerful remote system at a supercomputer center could be considered to fall into this category. In the other, a single node of the parallel computer is directly connected to a display. For example, a user with a small local cluster might fall into this category. The frontend-backend division of Neurondiz makes it relatively easy to develop code for these two setups. In the first configuration, the frontend and backend run on separate computers and they communicate using sockets. In the second configuration, the frontend and backend reside on the same system and their communication depends on which frontend is being used. The C++ frontend can simply call functions defined in the C backend. On the other hand, the Java frontend can communicate with the C backend using sockets, but we expect that performance will be improved using Java's Native Interface, which allows a JVM to interoperate with programs written in other languages. At the time of this writing (May, 2003) we support direct communication with the C++ frontend and communication with sockets with the Java frontend.

When Neurondiz is started, the backend opens the data file(s) storing the results from ooNeuroSys, and some basic information on the neurons is sent to the frontend, which opens a display showing a rectangular grid of disks, one disk for each neuron. In the most basic execution of Neurondiz, the program provides an animated display of the firing of the individual neurons by changing the color of the disks. The user can either single-step through the simulation by using mouseclicks, or she can let the animation proceed without user interaction.

If, at any time, the user wants more detailed information, she can select a rectangular subset of neurons and the subset will be displayed in a new window, which supports all the functionality of the original window. For even more detail, the user can select a single neuron and display its membrane voltage as a function of time.

The computational neuroscientists that used the original version of Neurondiz requested that the display also provide information on the synaptic connections among the neurons. Of course, the interconnection network for only a few hundred neurons, can contain tens-of-thousands of synaptic interconnections, which would be impossible to display with any clarity. So after some discussion with the neuroscientists, we developed two additional functionalities for Neurondiz. For the first, we added a functionality that allows a user to select a subset of neurons and display the interconnections among these neurons by interfacing with the graph visualization software package, Graphviz [3]. The display showing the neurons and their synaptic interconnections will, in general, require relocation of the neurons relative to each other in order to achieve a reasonably clear image. Thus, the selected neurons in the original display are keyed to the neurons in the Graphviz display by assigning matching numbers to paired neurons. At the time of this writing (May, 2003), the Graphviz interface is available only with the Java frontend. It is supplied by a GraphViz program called *Grappa*. Grappa sends a graph description to another Graphviz program called *dot,* and dot generates the layout and sends the layout back to Grappa for display.

The second functionality allows neuroscientists to determine the density of synaptic connections between two neurons. After the user selects two neurons, a source and a destination, and enters a parameter specifying the maximum path length, the backend will identify the subgraph consisting of all directed paths of synaptic connections from the source to the destination that consist of a number of synaptic connections less than or equal to the bound. The algorithm for finding the subgraph uses two breadth-first searches. Both Neurondiz and ooNeuroSys store synaptic interconnection information by storing lists of neurons with synaptic connections *into* each neuron assigned to a process. Hence the first breadth-first search works backwards from the target vertex. Essentially, the standard serial breadth-first algorithm is modified so that a globally replicated data structure records all visited vertices, and on each pass through the search loop, each process adds any vertices that can be reached from already visited vertices. At the end of the body of the search loop the structure recording visited vertices is updated. The loop terminates after it has searched the tree based at the destination with height equal to the user-specified bound.

At this point, we assume that the number of vertices visited by the breadth-first search is small enough so that the subgraph spanned by its neurons can be contained in the memory of a single process. So after the breadth-first search is completed, each process allocates enough storage for an adjacency matrix on the visited vertices. Each process initializes those columns of the matrix about which it has information, and a global reduction using logical-or creates a copy of the adjacency matrix on each process. The algorithm is completed by a breadth-first-search from the source on the subgraph induced by the vertices visited in the first search. Any unvisited vertices and edges not lying on paths with length less than or equal to the the bound are deleted.

5 Python Interface

High speed and parallelism are certainly required for a high fidelity simulation. However, robust, accurate, and innovative models are also necessary to achieve understanding of neural processes. Herein lies one of the principle problems with the development of object-oriented NeuroSys. High speed and parallelism requirements decrease programmability, especially for non-computer scientists. This is discouraging, particularly for smaller runs in which a scientist is trying to develop new models. To require neuroscientists to write code in C and to compile and link a new equation model is too great a burden. In ooNeuroSys we provide an interface from the core simulation to the Python Interpreted language [13] so that users can develop and run new equation models and manipulate the neural simulation directly.

The most important considerations in the design of the Python interfaces are simplicity and flexibility. For instance, the equations that model a simulation variable should be simple to program and extend even for a non-programmer:

```
def computeHPrime(self,h,V):
    def ah(v): return (0.128*exp(-(v+50.0)/18.0))
    def bh(v): return (4.0/(1.0+exp(-(v+27.0)/5.0)))

    h_prime = ah(V)*(1-h)-bh(V)*h

    return h_prime
```

The user should be able to examine and change state variables and connectivity:

```
for neuron in net:
    print 'my id is',neuron.id
    print 'my adjacency_list is',neuron.adjacency_list
    for neighbor in neuron.adjacency_list:
        print 'connect',neuron.id,'with',neighbor
    print 'voltage',neuron.V
```

Using these techniques, a user can setup and manipulate (steer) all relevant parts of a simulation. Interpreted languages may be slower than compiled languages like C, but Python of the sort used in the equations can be automatically compiled into high performance code [5, 8].

6 Performance

At the time of this writing (May 2003), we have only been able to run a few small tests. However, these tests suggest that ooNeuroSys is both faster and more scalable than the original NeuroSys. Table 6 shows the run times in seconds of some simulations run on the Keck Cluster [6], a Myrinet connected cluster of 64 dual processor Pentium III's. The systems being simulated use two neuron models (excitatory and inhibitory neurons), and the synaptic interconnection network was randomly generated. Each neuron is modelled with 5 variables. The dashes in the table either correspond to data which took too long to generate or a breakdown in the scalability of the program.

Table 1. Run Times of NeuroSys (Neuro) and ooNeuroSys (ooNeur)

	Neurons									
	16,384		32,768		65,536		131,072		262,144	
Processes	Neuro	ooNeur	Neuro	ooNeur	Neuro	ooNeur	Neuro	ooNeur	Neuro	ooNeur
1	544	372	1083	718	2168	1453	4295	—	—	—
2	277	239	543	463	1109	981	2220	1781	—	—
4	140	117	275	237	561	493	1743	950	2237	1930
8	73	51	142	119	288	239	581	483	1389	978
16	40	22	76	52	152	123	464	248	604	495
32	—	11	48	24	—	57	460	125	343	258

7 Conclusions and Directions for Future Work

Object-oriented NeuroSys provides many improvements over its predecessor. Its design provides both ease of maintenance and high performance. Preliminary tests indicate that it is both faster and more scalable than the original NeuroSys. The Python interface to ooNeuroSys promises to make it much easier to use. The new version of Neurondiz has a much improved interface and considerably more functionality.

Among other improvements to ooNeuroSys, we plan to make use of graph partitioning software to further improve scalability. We have already begun work on improving the scalability of I/O by incorporating some of the recommendations in [1]. By using the results of this work we hope to be able to develop a scalable I/O scheme that is well-suited to clusters that don't have parallel file

systems. Taken together, the Java and C++ versions of Neurondiz have all the desired functionality. However, both need to be completed.

References

1. Gregory Benson, Kai Long, Peter Pacheco: "The measured performance of parallel disk write methods on Linux multiprocessor nodes." To appear.
2. George D. Byrne and Alan C. Hindmarsh, "Correspondence: PVODE, an ODE solver for parallel computers," *The International Journal of High Performance Computing Applications,* vol 13, no 4, 1999, pp 354–365.
3. Emden R. Gansner and Stephen C. North, "An open graph visualization system and its applications to software engineering," *Software—Practice and Experience,* vol 30, no 11, pp 1203–1233, 1999.
4. A.L. Hodgkin and A.F. Huxley, "A quantitative description of membrane current and its application to conduction and excitation in nerve," *Journal of Physiology* (London), vol. 117, pp. 500–544, 1952.
5. Eric Jones and Patrick J. Miller, "Weave – inlining C/C++ in Python," in *O'Reilly Open Source Convention*, July 2002, `http://conferences.oreillynet.com/cs/os2002/view/e_sess/2919`.
6. *The Keck Cluster,* `http://keckclus.cs.usfca.edu/.`
7. *LAM/MPI Parallel Computing,* `http://www.lam-mpi.org/.`
8. Patrick J. Miller, " Parallel, Distributed Scripting with Python," in *Third International Conference on Linux Clusters: The HPC Revolution,* St. Petersburg, FL, Oct 23 - Oct 25, 2002, `http://www.linuxclustersinstitute.org/Linux-HPC-Revolution/Archive/2002techpapers.html`.
9. *MPICH-A Portable Implementation of MPI,* `http://www-unix.mcs.anl.gov/mpi/mpich/index.html.`
10. *Myrinet Software and Documentation,* `http://www.myri.com/scs/.`
11. Peter Pacheco, Marcelo Camperi, and Toshiyuki Uchino: "Parallel NeuroSys: a system for the simulation of very large networks of biologically accurate neurons on parallel computers," *Neurocomputing,* vol 32–33, 2000, pp. 1095-1102.
12. *Parallel NeuroSys,* `http://www.cs.usfca.edu/neurosys`.
13. *The Python Language,* `http://www.python.org`.
14. Trolltech, *Qt 3.1 Whitepaper,* `www.trolltech.com.`
15. Ray S. Tuminaro, John N. Shadid, and Scott A. Hutchinson, "Parallel Sparse Matrix-Vector Multiply Software for Matrices with Data Locality," Sandia National Laboratories, SAND95-1540J, 1995.
16. Mason Woo, Jackie Neider, Tom Davis, and Dave Shreiner, *OpenGL(R) Programming Guide: The Official Guide to Learning OpenGL, Version 1.2,* 3rd ed., Addison-Wesley Publishing Co, Boston, MA, 1999.